ANALYSIS OF A TABLE-DRIVEN ALGORITHM FOR FAST CONTEXT-FREE PARSING

James R. Kipps

June 1988

P-7452

The RAND Corporation, 1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

PREFACE

The work reported here began in connection with the ROSIE® Language Development Project, which was funded by the Information Sciences and Technologies Office of the Defense Advanced Research Projects Agency under the auspices of The RAND Corporation's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Since the completion of the ROSIE project, this work was continued independently by the author. This Paper is a companion to the RAND Note "A TABLE-DRIVEN APPROACH TO FAST CONTEXT-FREE PARSING" (N-2841-DARPA). The intention in publishing both documents is to make the parsing techniques applied in ROSIE available for use by others at RAND and elsewhere.

SUMMARY

A variation on Tomita's algorithm for general context-free parsing is analyzed in regards to its time complexity. It is shown to have a general time bound proportional to $n^{\bar{\rho}+1}$, where n is the length of the input string and $\bar{\rho}$ is the length of the longest production in the source grammar. A modification of this algorithm is presented for which this time bound is reduced to a factor of n^3. A discussion of space bounds, as well as two subclasses of context-free grammars that can be recognized in less than $O(n^3)$ time, is also included.

CONTENTS

FIGURES

1. INTRODUCTION

Context-free grammars (Chomsky, 1956) have been widely used in describing the syntax of programming and natural languages. Numerous algorithms have been developed to recognize sentences in languages so described. Some are general, in the sense that they can handle all or most context-free grammars; others are more restricted and can handle only a small subclass of these grammars (including the grammars of most programming languages). These latter algorithms, e.g., the LL, operator precedence, predictive, and LR parsing algorithms (Aho and Ullman, 1972) are typically more efficient than the former because they take advantage of inherent features in the class of grammars they recognize.

Most practical parsers analyze the syntax of their input in a single deterministic pass, without need of backup. Each symbol is examined only once, and, at the time it is examined, there is sufficient information available to make any necessary parsing decisions. In his famous paper, Knuth (Knuth, 1965) established a family of context-free grammars known as *LR(k)* grammars and provided an effective test to determine, for a given positive integer k, whether a grammar belonged to the LR(k) class. The connection to practical parsers mentioned above is that an LR(k) grammar describes a language, all of whose sentences can be parsed in a single backup-free parse, if at most k symbols of look-ahead are available. Despite the wide coverage of LR(k) grammars, there are still many languages for which no LR(k) grammar exists. The example that sparked this work is ROSIE (Kipps et al., 1987), a language for applications in artificial intelligence with a high-level English-like syntax.

The general context-free parsing algorithms, e.g., Earley's algorithm (Earley, 1968; 1970), and the Cocke-Younger-Kasami algorithm (Younger, 1967), must necessarily simulate a nondeterministic pass over their input, using some form of search. Due to the inefficiency this causes, these algorithms have not been widely used as practical parsers for programming languages. However, as programming languages, like ROSIE, begin to approach natural language in readability and expressiveness, they will also become ambiguous and non-LR(k) and, thus, inherently harder to recognize. Such grammars require the power of a general parser. Although the best time bound for the general context-free algorithms is $O(n^3)$,[2] where n is the length of the input string, some of these algorithms run faster for certain subclasses of context-free grammars. For instance, in his thesis (Earley, 1968), Earley shows

[2] Actually, the best upper bound is Valiant's algorithm, which runs in $O(n^{2.81})$. However, since this is also its lower bound, Valiant's algorithm is only of theoretical interest.

how his algorithm runs in $O(n^2)$ for unambiguous grammars and in time $O(n)$ for *bounded state* grammars. Unambiguous and bounded state grammars subsume the LR(k) grammars, and they may also subsume a large subset of interesting non-LR(k) grammars. Although Earley's algorithm is still too inefficient to be used as a practical parser for even these grammars, their existence suggests that a sufficiently fast algorithm for practical general context-free parsing could be developed.

A basic characteristic that seems to be shared by the best known of the general context-free algorithms is that they are top-down parsers. Recently, however, Tomita (Tomita, 1985a;b) introduced an algorithm (intended for natural language applications) that takes advantage of an extended LR parse table. This would appear to be the first general context-free algorithm that is a bottom-up parser. The obvious benefit of this approach is that it eliminates the need to expand alternatives of a nonterminal at parse time (i.e., what Earley calls the *predictor* operation). For Earley's algorithm, eliminating the predictor operation would not change its upper-bound time complexity, but it would save a factor of n^2, which in itself could be significant to a practical parser. Unfortunately, upon examination Tomita's algorithm is found to have a general time complexity of $O(n^{\bar{\rho}+1})$, where n is as before and $\bar{\rho}$ is the length of the longest production in the source grammar. Thus, this algorithm achieves $O(n^3)$ for grammars in Chomsky normal form (Chomsky, 1959) but has potential for being worse in unrestricted grammars.

In this paper, I present a modification of Tomita's algorithm that allows it to run in time proportional to n^3 for grammars with any length productions. First, in Section 2, the terminology used in this paper is defined. A variantion on Tomita's algorithm is described informally as a recognizer in Section 3, and formally defined and analyzed in Section 4. Section 5 describes and analyzes the modification to this algorithm that allows it to run in time $O(n^3)$. Section 6 examines the cost in terms of space required by the modified algorithm. Finally, Section 7 describes two subclasses of context-free grammars for which the algorithm runs in $O(n^2)$ and $O(n)$, respectively.

2. TERMINOLOGY

A *language* is a set of strings over a finite set of symbols. These symbols are called *terminals* and are represented by lowercase letters, e.g., a, b, c. A *context-free grammar* is used as a formal device for specifying which strings are in a language; hereafter, grammar is used to mean context-free grammar. A grammar uses another set of symbols called *nonterminals*, which define the syntactic classes of the language; nonterminals are represented by capital letters, e.g., A, B, C. Together the terminals and nonterminals of a language make up its *vocabulary*. Strings of vocabulary symbols are represented by Greek letters, e.g., α, β, γ. The empty string is ϵ. $|\alpha|$ is the number of symbols in α.

A grammar consists of a finite set of rewrite rules or *productions* of the form

$$\mathtt{A} \rightarrow \alpha$$

where the '$\mathtt{A}$' component is called the *left-hand side* of the production, and the 'α' component is called its *right-hand side*. The nonterminal that stands for "sentence" is called the *root* (R) of the grammar. Productions with the same nonterminal on their left-hand side are called *alternatives* of that nonterminal. Productions of the form

$$\mathtt{A} \rightarrow \epsilon$$

are called *null productions*.

The rest of the definitions are given with respect to a particular *source* grammar G. We write

$$\alpha \Rightarrow \beta$$

if $\exists \gamma, \delta, \eta, \mathtt{A}$ such that $\alpha = \gamma\mathtt{A}\delta$ and $\beta = \gamma\eta\delta$ and $\mathtt{A} \rightarrow \eta$ is a production. We write

$$\alpha \overset{*}{\Rightarrow} \beta$$

(β is *derived* from α) if $\exists \alpha_0, \alpha_1, \cdots, \alpha_m$ $(m \geq 0)$ such that

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_m = \beta.$$

The sequence $\alpha_0, \cdots, \alpha_m$ is called a derivation (of β from α).

A *sentential form* is a string α such that the root $R \overset{*}{\Rightarrow} \alpha$. A *sentence* is a sentential form consisting entirely of terminal symbols. The *language defined by a grammar L(G)* is the set of its sentences. Any sentential form may be represented in at least one way as a *derivation tree*, reflecting the steps made in deriving it (though not the order of the steps). The *degree of ambiguity* of a sentence is the number of its distinct derivation trees. A sentence is *unambiguous* if it has degree 1 of ambiguity. A grammar is *unambiguous* if each of its sentences is unambiguous.

A *recognizer* is an algorithm that takes as its input a string and either *accepts* or *rejects* it, depending on whether the string is a sentence of the language defined by the grammar. A *parser* is a recognizer that outputs the set of all legal derivation trees of a string upon acceptance.

3. TOMITA'S ALGORITHM

The following is an informal description of a variation on Tomita's algorithm as a recognizer. I assume familiarity with standard LR parsing, the exact definition and operation of which can be found in Aho and Ullman, 1972. The changes introduced to Tomita's algorithm do not alter it significantly, but they do make it easier to describe and analyze. The algorithm described below is based on an implementation of Tomita's algorithm as a practical parser (Kipps, 1988) distributed with the ROSIE language.

Tomita views his algorithm as a variation on standard LR parsing. The algorithm takes a shift-reduce approach, using an extended LR parse table to guide its actions. The changes the algorithm makes to the parse table are to allow it to contain multiple actions per entry, i.e., at most one *shift* action or *accept* action and any number of *reduce* actions. Thus, the parse table can no longer be used for strictly deterministic parsing; some search must be done. The algorithm emulates a nondeterministic parse with pseudo-parallelism. It scans an input string $x_1 \cdots x_n$ from left to right, following all paths in a breath-first manner and merging like subpaths when possible to avoid redundant computations.

An example non-LR grammar is shown in Figure 3.1,

```
(1)   S  → NP VP
(2)   S  → S PP
(3)  NP  → *n
(4)  NP  → *det *n
(5)  NP  → NP PP
(6)  PP  → *prep NP
(7)  VP  → *v NP
```

Fig. 3.1—Example of Non-LR Grammar

and its parse table in Figure 3.2. In building the parser table the grammar is augmented by a 0th production

$$D_0 \rightarrow R \dashv$$

where R is the root of the grammar and where the symbol '$\dashv$' is a special terminal that denotes end-of-sentence and appears only as the last symbol of an input string. Entries '**sh** *s*' in the *action table* (the left part of the table) indicate the action '*shift to State s.*' Entries '**re** *p*' indicate the action '*reduce constituents on the stack according to Production p.*' The entry '**acc**' indicates the action '*accept,*' and blank spaces represent '*error.*' Entries '*s*' in the *goto table* (the right part of the table) indicate the action '*after a reduce action, shift to State s.*' In Figure 3.2, there are two

multi-action entries in States 3 and 11 under the column labeled '*prep.'

State	*det	*n	*v	*prep	⊣	NP	PP	VP	S
0	sh5	sh7				9			1
1				sh2	acc		8		
2	sh5	sh7				3			
3			re6	re6,sh2	re6		4		
4			re5	re5	re5				
5		sh6							
6			re4	re4	re4				
7			re3	re3	re3				
8				re2	re2				
9			sh10	sh2			4	12	
10	sh5	sh7				11			
11				re7,sh2	re7		4		
12				re1	re1				

Fig. 3.2—LR Parse Table with Multiple Entries

The algorithm operates by maintaining a number of *processes* in parallel. Each process has a stack and behaves basically the same as in standard LR parsing. Each stack element is labeled with a parse state and points to its parent, i.e., the previous element on a process's stack. We call the top-of-stack the *current state* of a process.

Each process does not actually maintain its own separate stack. Rather, these "multiple" stacks are represented using a single directed acyclic (but reentrant) graph called a ***graph-structured stack***. Each stack element is a vertex of the graph. Each leaf of the graph acts as a distinct top-of-stack to a process, while the root of the graph acts as a common bottom-of-stack. The edge between a vertex and its parent is directed toward the parent. Because of the reentrant nature of the graph, a vertex may have more than one parent.

The leaves of the graph grow in stages; each stage U_i corresponds to the ith symbol x_i from the input string. After x_i is scanned, the leaves in stage U_i are in a one-to-one correspondence with the algorithm's *active* processes, where each process references a distinct leaf and treats that leaf as its current state. Upon scanning x_{i+1}, an active process can either (1) add another leaf to U_i, or (2) add a new leaf to U_{i+1}. Only processes that added leaves to U_{i+1} will be active when x_{i+2} is scanned.

In general, a process behaves in the following manner. On x_i, each active process (corresponding to the leaves in U_{i-1}) looks up and executes the entries in the action table for x_i given its current state. When a process encounters multiple actions, it *splits* into several processes (one for each action), each sharing a common top-of-stack. When a process encounters an error entry, the process is *discarded* (i.e., its top-of-stack vertex sprouts no leaves into U_i by way of that process). All processes are synchronized, scanning the same symbol at the same time. Thus, after a process shifts

on x_i into U_i, it waits until there is no other processes that can act on x_i before scanning x_{i+1}.

The Shift Action. A process (with top-of-stack vertex v) shifts on x_i from its current state s to some successor state s' by

(1) creating a new leaf v' in U_i labeled s';

(2) placing an edge from v' to its top-of-stack v (directed towards v); and

(3) making v' its new top-of-stack vertex (in this way changing its current state).

Any successive process shifting to the same state s' in U_i is *merged* with the existing process to form a single process whose top-of-stack vertex has multiple parents, i.e., by placing an additional edge from the top-of-stack vertex of the existing process in U_i to the top-of-stack vertex of the shifting process. The merge is done because, individually, these processes would behave in exactly the same manner until a reduce action removed the vertices labeled s' from their stacks. Thus, merging avoids redundant computation. Merging also insures that each leaf in any U_i will be labeled with a distinct parse state, which puts a finite upper-bound on the possible number of active processes and, thus, the size of the graph-structured stack.

The Reduce Action. A process executes a reduce action on a production p by following the chain of parent links down from its top-of-stack vertex v to the ancestor vertex from which the process began scanning for p earlier, essentially "popping" intervening vertices off its stack. Since merging means a vertex can have multiple parents, the reduce operation can lead back to multiple ancestors. When this happens, the process is again split into separate processes (one for each ancestor). The ancestors will correspond to the set of vertices at a distance $\bar{p}$ from v, where $\bar{p}$ equals the number of symbols in the right-hand side of the pth production. Once reduced to an ancestor, a process shifts to the state s' indicated in the goto table for D_p (the nonterminal on the left-hand side of the pth production) given the ancestor's state. A process shifts on a nonterminal much as it does a terminal, with the exception that the new leaf is added to U_{i-1} rather than U_i. (A process can only enter U_i by shifting on x_i.)

The algorithm begins with a single initial process whose top-of-stack vertex is the root of the graph-structured stack. It then follows the general procedure outlined above for each symbol in the input string, continuing until there are either no leaves added to U_i (i.e., no more active processes), which denotes *rejection*, or a process executes the accept action on scanning the $n+1$st input symbol '$\dashv$,' which denotes *acceptance*.

4. ANALYSIS OF TOMITA'S ALGORITHM

In this section, I present a formal definition of the variation on Tomita's algorithm described in Section 3, as a recognizer for an input string $x_1 \cdots x_n$ and then analyze the time complexity of the algorithm according to this definition. This definition is understood to be with respect to an extended LR parse table (with start state S_0) constructed from a source grammar G.

Notation. Number the productions of G arbitrarily $1, \cdots, d$, where each production is of the form

$$D_p \rightarrow C_{p1} \cdots C_{p\bar{p}} \quad (1 \leq p \leq d)$$

and where $\bar{p}$ is the number of symbols on the right-hand side of the pth production.

Definition. The entries of the extended LR parse table are accessed with the functions ACTIONS and GOTO.

- ACTIONS(s,x) returns a set of actions from the action table along the row of State s under the column labeled 'x.' This set will contain no more than one of a shift action '**sh** s'' *or* an accept action '**acc**;' it may contain any number of reduce actions '**re** p.'
- GOTO(s,D_p) returns a state 's'' from the goto table along the row of State s under the column labeled with nonterminal D_p.

Definition. Each *vertex* of the graph-structured stack is a triple $\langle i, s, l \rangle$, where i is an integer corresponding to ith input symbol scanned (i.e., the time at which the vertex was created as a leaf), s is a parse state (corresponding to a row of the parse table), and l is a set of parent vertices. The *processes* described in the last section are represented implicitly by the vertices in successive U_i's. The root of the graph-structured stack, and hence the initial process, is the vertex $\langle 0, S_0, \{\} \rangle$.

The Recognizer. The recognizer is a function of one argument REC($x_1 \cdots x_n$). It calls upon the functions SHIFT(v,s) and REDUCE(v,p). SHIFT(v,s) either (1) adds a new leaf to U_i labeled with parse state s whose parent is vertex v or (2) merges vertex v with the parents of an existing leaf; REDUCE(v,p) executes a reduce action from vertex v using production p. REDUCE calls upon the function ANCESTORS(v,$\bar{p}$), which returns the set of all ancestor vertices a distance of $\bar{p}$ from vertex v. These function are defined in Figure 4.1.

The definitions in Figure 4.1 vary somewhat from the formal definition given in (Tomita, 1985a).[3] As a brief explanation, in REC, [1] adds the end-of-sentence symbol '⊣' to the end of

[3] The changes introduced to Tomita's algorithm do not alter it significantly, but they do make it easier to describe. In particular, Tomita's functions REDUCE and REDUCE-E have been collapsed into a single function.

the input string; [2] initializes the root of the graph-structured stack; [3] iterates through the symbols of the input string. On each symbol x_i, [4] processes the vertices (denoting the active processes)

```
     REC(x_1 ··· x_n)
[1]   Let x_{n+1} := ⊣
      Let U_i be empty    (0 ≤ i ≤ n)
[2]   Let U_0 := {⟨0,S_0,{}⟩}
[3]   For i from 1 to n+1
        Let P be empty
[4]     For each v = ⟨i-1,s,l⟩ ∈ U_{i-1},
          Let P := P ∪ {v}
[5]       If 'sh s'' ∈ ACTIONS(s,x_i), SHIFT(v,s')
[6]       For each 're p' ∈ ACTIONS(s,x_i), REDUCE(v,p)
[7]       If 'acc' ∈ ACTIONS(s,x_i), accept
[8]     If U_i is empty, reject

     SHIFT(v,s)
[9]   If ~∃⟨i,s,l⟩ ∈ U_i,
        let U_i := U_i ∪ {⟨i,s,{v}⟩}
      else
        let l := l ∪ {v}

     REDUCE(v,p)
[10]  For each v' = ⟨j',s',l_1'⟩ ∈ ANCESTORS(v,p̄),
        Let s'' := GOTO(s',D_p)
[11]    If ~∃v'' = ⟨i-1,s'',l''⟩ ∈ U_{i-1},
          let U_{i-1} := U_{i-1} ∪ {⟨i-1,s'',{v'}⟩}
        else
[12]    If v' ∉ l'',
[13]      if ~∃⟨j',s',l_2'⟩ ∈ l'',
            let l'' := l'' ∪ {v'}
[14]        if v'' ∈ P,
              let v_d := ⟨i-1,s'',{v'}⟩
              for each 're p' ∈ ACTIONS(s'',x_i),
                REDUCE(v_d,p)
        else
          do nothing (ambiguous)

     ANCESTORS(v = ⟨j,s,l⟩,c)
[15]  If c = 0,
        return({v})
      else
        return(⋃_{v'∈l} ANCESTORS(v',c-1))
```

Fig. 4.1—Tomita's Algorithm

of successive U_{i-1}'s, adding each vertex to P to signify that it has been processed; [5], [6] and [7] respectively execute the shift, reduce and accept actions from the action table given the state s of a vertex; and [8] checks that at least one vertex was added to U_i, insuring that at least one process is still active after processing x_i and before scanning x_{i+1}.

In SHIFT, [9] adds a vertex to U_i labeled s (i.e., shifts a process to state s in U_i). If a vertex labeled s does not already exist, one is created with a single parent v; otherwise, v is added to the parents of the existing vertex, thus merging processes.

In REDUCE, [10] iterates through the ancestor vertices a distance of $\bar{p}$ from v, setting s'' to the state indicated in the goto table on D_p given the state of the ancestor. Each ancestor vertex is shifted into U_{i-1} on s'': [11] adds a vertex labeled s'' to U_{i-1} if no such vertex already exists. If such a vertex does exist, [12] checks that a shift from the current ancestor v' has not already been made. (If it has, then some segment of the input string has been recognized as an instance of the same nonterminal D_p in two different ways, and the current derivation can be discarded as ambiguous; otherwise, v' is merged with the parents of the existing vertex.) Before merging, [13] checks that v' is not a "dummy" vertex, created by [14] from an earlier call to REDUCE; [14] checks if the vertex v'' has already been processed. If so, then it missed any possible reductions through v', so a dummy vertex v_d is created as a variant on v'' with a single parent v'. For all such reduce actions, REDUCE is called using v_d in place of v''. Because of reduce actions on null productions, it is possible for ANCESTORS to return a dummy vertex as the ancestor of itself; so, going back to [13], if a variant of v' already exists in the parents of v'', then v' is a dummy vertex and can be discarded; otherwise, it is a real vertex and can be added to the parents of v''.

Finally, in ANCESTORS, [15] recursively descends the chain of parents of vertex v, returning the set of vertices a distance of c from v.

The changes introduced in Figure 4.1 do not alter Tomita's algorithm significantly, but they do make it easier to develop some ideas about its efficiency. Here we wish to find the upper bounds on time as a function of n (the length of the input string). Since there are existing general context-free parsing algorithms that are $O(n^3)$ with respect to time, it is of interest to analyze Tomita's algorithm and see how it compares.

The General Case. Tomita's algorithm is an $O(n^{\bar{p}+1})$ recognizer in general, where $\bar{p}$ is the greatest $\bar{p}$ in G. The reasons for this are:

(a) Since each vertex in U_i must be labeled with a distinct parse state, the number of vertices in any U_i is bounded by the number of parse states;

(b) The number of parents l of a vertex $v = \langle i, s, l \rangle$ in U_i is proportional to i ($\sim i$). This is because a process could have started scanning for a production p in each U_j ($j \leq i$) and, thus, a process could reduce on p in U_i and split into to $\sim i$ processes (one for each ancestor in a distinct U_j). Each process could shift on D_p to the same state in U_i and, thus, that vertex could have $\sim i$ parents;

(c) For each x_{i+1}, SHIFT will be called a bounded number of times, i.e., at most once for each

vertex in U_i; SHIFT executes in a bounded number of steps.

(d) For each x_{i+1} and production p, REDUCE(v,p) will be called a bounded number of times in REC, and REDUCE(v_d,p) (the recursive call to REDUCE) will be called no more than $\sim i$ times. The reason for the former is the same as in (c), while the latter is due to the conditions on the recursive call, which maintain that it can be called no more than once for each parent of a vertex in U_i, of which there are at most $\sim i$;

(e) REDUCE(v,p), because at most $\sim i$ vertices can be returned by ANCESTORS, executes in $\sim i$ steps plus the steps needed to execute ANCESTORS.

(f) ANCESTORS(v,$\bar{p}$) executes in $\sim i^{\bar{p}}$ steps in the worst case. This is because, while at most $\sim i$ processes could have started scanning for p, the number of paths by which any single process could reach v in U_i is dependent upon the number of ways the intervening input symbols can be partitioned among the $\bar{p}$ vocabulary symbols in the right-hand side of production p. For a process that started from U_j ($j \leq i$), the number of paths to v in U_i in the recognition of p can be proportional to

$$\sum_{m_1=j}^{0} \sum_{m_2=m_1}^{0} \cdots \sum_{m_{\bar{p}-1}=m_{\bar{p}-2}}^{0} 1.$$

Summing from $j = 0, \cdots, i$ gives the closed form $\sim i^{\bar{p}}$. ANCESTORS($v_d = \langle i, s\{v'\}\rangle$,$\bar{p}$) executes in $\sim i^{\bar{p}-1}$ steps because there is that many ways $\sim i$ ancestor vertices could reach v', and only one way v' could reach v;

(g) The worst case time bound is dominated by the time spent in ANCESTORS, which can be added to the time spent in REDUCE. Since REDUCE(v,p), with a bound $\sim i^{\bar{p}}$, is called only a bounded number of times, and REDUCE(v_d,p), with a time bound of $\sim i^{\bar{p}-1}$, is called at most $\sim i$ times, the worst case time to process any x_i is $\sim i^{\bar{p}}$, for each $i = 0, \cdots, n+1$ and longest production ρ;

(h) Summing from $i = 0, \cdots, n+1$ gives REC a general time bound $\sim i^{\bar{p}+1}$.

This bound indicates that Tomita's algorithm belongs to complexity class $O(n^3)$ only if applied to grammars in Chomsky normal form (CNF)[3] or some other equally restricted notation. Although any context-free grammar can be automatically converted to CNF (Hopcraft and Ullman, 1979), extracting useful information from the derivation trees produced by such grammars would be time consuming at best (if possible at all).

[3] In CNF, productions can have one of two forms, A → BC or A → a; thus, the length of the longest production is at most 2.

5. MODIFYING THE ALGORITHM FOR N^3 TIME

In this section, I explain how to turn this algorithm into an n^3 recognizer for context-free grammars in any unrestricted form. ANCESTORS is the only function which forces us to use $i^{\bar{p}}$ steps. It is interesting to note, however, that ANCESTORS can take this many steps even though it returns at most $\sim i$ ancestor vertices, and even though there are at most $\sim i$ intervening vertices and edges between a vertex in U_i and its ancestors. This points out the fact that ANCESTORS follows the same subpaths more than once. The efficiency of ANCESTORS can be greatly improved if this redundancy is eliminated.

The modification described here turns ANCESTORS into a table look-up function. That is, assume that there is a two-dimensional "ancestors" table. One dimension is indexed on the vertices in the graph-structured stack, and the other is indexed on integers $c = 1$ to $\bar{\rho}$, where $\bar{\rho}$ equals the greatest $\bar{p}$. Each entry (v, c) is the set of ancestor vertices a distance of c from vertex v. Then, ANCESTORS(v, c) returns the (at most) $\sim n$ ancestor at (v, c) in ~ 1 steps. Of course, the table must be filled dynamically during the recognition process, and so we must calculate the time expended in this task.

In Figure 5.1, ANCESTORS is defined as a table look-up function that dynamically generates table entries the first time they are requested. In this definition, the "ancestor" table is represented by changing the parent field l of a vertex $v = \langle i, s, l \rangle$ from a set of parent vertices to an ancestor field a. For a vertex $v = \langle i, s, a \rangle$, a consists of a set of tuples $\langle c, l_c \rangle$, such that l_c is the set of ancestor vertices a distance of c from v. Thus, the portion of entries for each vertex in the ancestor table is associated with the vertex itself.[4]

Figure 5.1 illustrates the necessary modifications made to the definitions of Figure 4.1. (No changes are made to REC.) In SHIFT, [1] adds a vertex to U_i labeled s. If such a vertex does not already exist, one is created whose ancestor field records that v is the ancestor vertex at a distance of 1; otherwise, v is added those ancestors.

In REDUCE, [2] iterates through the ancestors v' a distance of $\bar{p}$ from v, finding the appropriate state s'' to shift to from the goto table; [3] adds a vertex labeled s'' to U_{i-1} (if no such vertex already exists) whose ancestors field is initialized to point back to v'. if such a vertex does exist, [4] checks for an ambiguity. if there is no ambiguity, then v' is merged with the other ancestors a distance of 1

[4] While this definition may at first seem obtuse, it was adopted to suggest an implementation that could take advantage of the LISP garbage collector to dynamically recover table entries when a process is terminated.

from v''. [5] first checks that v' is not a dummy vertex (as described in Section 4) created by [6] in an earlier call to REDUCE; [6] checks if v'' has already been processed. If so, it applies REDUCE to

```
     SHIFT(v,s)
[1]   If ~∃⟨i,s,a⟩ ∈ U_i,
        let U_i := U_i ∪ {⟨i,s,{⟨1,{v}⟩}⟩}
      else
        let l_1 := l_1 ∪ {v} | ⟨1,l_1⟩ ∈ a

     REDUCE(v,p)
[2]   For each v' = ⟨j',s',a_1'⟩ ∈ ANCESTORS(v,p̄),
        Let s'' := GOTO(s',D_p)
[3]     If ~∃v'' = ⟨i-1,s'',a_1''⟩ ∈ U_{i-1},
          let U_{i-1} := U_{i-1} ∪ {⟨i-1,s'',{⟨1,{v'}⟩}⟩}
        else
[4]     If v' ∉ l_1 | ∃⟨1,l_1⟩ ∈ a_1'',
[5]       if ~∃⟨j',s',a_2'⟩ ∈ l,
            let l_1 := l_1 ∪ {v'}
[6]         if v'' ∈ P,
              let v_d := ⟨i-1,s'',a_2'' = {⟨1,{v'}⟩}⟩
              for each 're p' ∈ ACTIONS(s'', x_i),
                REDUCE(v_d,p)
[7]           for each ⟨c,l_{c_2}⟩ ∈ a_2'' | c ≥ 2,
                if ∃⟨c,l_{c_1}⟩ ∈ a_1''
                  let l_{c_1} := l_{c_1} ∪ l_{c_2}
                else
                  let a_1'' := a_1'' ∪ {⟨c,l_{c_2}⟩}
        else
          do nothing (ambiguous)

     ANCESTORS(v = ⟨j,s,a⟩,c)
[8]   If c = 0,
        return({v})
      else
      If ∃⟨c,l_c⟩ ∈ a,
        return(l_c)
      else
[9]     let l_c := ⋃_{v'∈l_1|⟨1,l_1⟩∈a} ANCESTORS(v',c-1)
        let a := a ∪ {⟨c,l_c⟩}
        return(l_c)
```

Fig. 5.1—Modified Algorithm[5]

v_d (a dummy v'') for each reduce action on x_i. After the application of REDUCE, [7] updates the ancestor table stored in v'' to record entries made in the ancestor field a_2'' of the dummy vertex.

In ANCESTORS, [8] looks up in a (the portion of the ancestor table stored with v) those vertices at a distance of c; if an entry exists, those vertices are returned, but if not [9] calls ANCESTORS

[5] The definition of REC from Figure 4.1 is unchanged.

recursively to generated those vertices and, before returning the generated vertices, records them in the ancestor field of v.

The question now becomes how much time is spent filling the ancestor table. For ANCESTORS(v,$\bar{p}$), this is bounded in the worst case by $\sim i^2$ steps, and for ANCESTORS(v_d,$\bar{p}$), it is bounded by $\sim i$ steps. This is because, in general, ANCESTORS($v = \langle i, s, a\rangle$,$c$) will take $\sim i$ steps to execute the first time it is called (one for each recursive call to ANCESTORS(v',$c-1$) ($v' \in l_1$ and $\langle 1, l_1\rangle \in a$)), plus the steps in the recursive call, and ~ 1 steps thereafter. When ANCESTORS(v,$\bar{p}$) is executed, there are $\sim i$ such "virgin" vertices between v (in U_i) and its ancestors, and so this call can execute $\sim i^2$ steps in the worst case. ANCESTORS(v_d,$\bar{p}$) is called only after the call to ANCESTORS(v,$\bar{p}$) has been made, and so $\sim i$ of the vertices between v' and the ancestor vertices have been processed; hence, the call to ANCESTORS(v',$\bar{p}-1$) could take $\sim i$ steps for each of a bounded number of intervening vertices.

Given this, the upper bound on the number of steps that can be executed by the total calls on REDUCE for a given x_i is $\sim i^2$. Summing from $i = 0, \cdots, n+1$ gives $\sim n^3$ steps as the worst case upper bound on the execution time of the modified algorithm.

6. SPACE BOUNDS

While it might be suggested that the modifications introduced in the last section make an unfavorable trade off between time and space, analysis actually shows that space efficiency is impaired by at most a constant factor.

The space complexity of Tomita's algorithm as it appears in Section 4 is proportional to n^2 in the worst case (ignoring the read-only space requirements for the parse table). This is because the space requirements of the algorithm are bounded by the requirements of the graph-structured stack. There are a bounded number of vertices in each U_i of the graph-structured stack, and each vertex can have at most $\sim i$ parents. Summing again from $i = 0, \cdots, n+1$ gives us $\sim n^2$ as the worst case space requirement for the graph-structured stack.

With the modification given in Section 5, we have increased the space requirements of the graph-structured stack by at most a constant factor of n^2. This is because the modification replaces the $\sim i$ parents of a vertex in U_i with at most $\sim \bar{\rho} i$ entries in the ancestors field. So, for a vertex $v = \langle i, s, a \rangle \in U_i$, the ancestors field a will be a subset of tuples $\langle c, l_c \rangle$ such that $1 \leq c \leq \bar{\rho}$ and $|l_c| \simeq i$. Summing from $i = 0, \cdots, n+1$, gives us $\sim \bar{\rho} n^2$ or $\sim n^2$ still as a worst case upper bound on space.

7. LESS THAN N^3 TIME

As mentioned earlier in the introduction, several of the better known general context-free algorithms have been shown to run in less than $O(n^3)$ time for certain subclasses of grammars. Therefore, it is of interest to ask if Tomita's algorithm, as well as the modified version presented here, can also recognize some subclasses of context-free grammars in less than $O(n^3)$ time. In this section, I informally describe two such subclasses that can be recognized in $O(n^2)$ and $O(n)$ time, respectively. The arguments for their existence parallel those given by Earley in his thesis (Earley, 1968), where they are formally specified.

Time $O(n^2)$ Grammars. ANCESTORS is the only function that forces us to use $\sim i^p$ steps in Tomita's algorithm and $\sim i^2$ steps in the modified algorithm. We determined that this could happen when a ancestor vertex v' from U_j $(j \leq i)$ reached the reducing vertex v in U_i by more than a single path, i.e., the symbols $x_j \cdots x_i$ were derived from a nonterminal D_p in more than one way, indicating that grammar G is ambiguous. If G were unambiguous, then there would be at most one path from a given v' to v. This means that the bounded calls to ANCESTORS(v,$\bar{p}$) can take at most $\sim i$ steps, and that ANCESTORS(v_d,$\bar{p}$) can take at most a bounded number of steps. The first observation is due to the fact that there are $\sim i$ ancestor vertices that can be reached in only one way. Similarly, the second is due to the fact that if ANCESTORS(v_d,$\bar{p}$) took $\sim i$ steps, returning $\sim i$ ancestors, and was called $\sim i$ times, then some ancestor vertices must have shifted into U_i in more than one way, which would be a contradiction, meaning grammar G must be ambiguous.

So, if the grammar is unambiguous, then the total time spent in REDUCE for any x_i is $\sim i$ and the worst case time bound for the Tomita's algorithm is $O(n^2)$. A similar result is true for the modified algorithm.

Time O(n) Grammars. In his thesis, Earley points out that "*... for some grammars the number of states in a state set can grow indefinitely with the length of the string being recognized. For some others there is a fixed bound on the size of any state set. We call the latter grammars bounded state grammars.*" While Earley's "states" have a different meaning than states in Tomita's algorithm, a similar phenomena occurs, i.e., for the bounded state grammars there is a fixed bound on the number of parents any vertex can have.

In Tomita's algorithm, bounded state grammars can be recognized in time $O(n)$ for the following reason. No vertex can have more than a bounded number of ancestors (if otherwise, then $\sim i$ vertices could be added to the parents of some vertex in U_i, proving by contradiction that the grammar is not bounded state). This then means that the ANCESTORS function can execute in a bounded number

of steps. Likewise, REDUCE can only be called a bounded number of times. Summing over the x_i gives us an upper bound $\sim n$. Again, a similar result is true for the modified algorithm. Interestingly enough, Earley states that almost all LR(k) grammars are bounded state, as well, which suggests that Tomita's algorithm, given k-symbol look ahead, should perform with little loss of efficiency as compared to a standard LR(k) algorithm when the grammar is "close" to LR(k). Earley also points out that not all bounded state grammars are unambiguous; thus, there are non-LR(k) grammars for which Tomita's algorithm can perform with LR(k) efficiency.

8. CONCLUSION

The results in this paper support in part Tomita's claim of efficiency for his algorithm. With the modification introduced here, Tomita's algorithm is shown to be in the same complexity class as existing general context-free algorithms. These results also give support to his claim that his algorithm should run with near LR(k) efficiency for near LR(k) grammars.

Because the algorithm is a bottom-up parser that takes advantage of the multi-action LR parse table, it avoids the extraneous computation associated with the goal expansion found in other general algorithms. This could eliminate a factor of n^2 steps, which in practical applications could result in significant gains in performance. The variation on Tomita's algorithm described in Section 4 and the modified algorithm have been implemented in LISP; the former implementation is distributed with the ROSIE programming language.

Although the modified algorithm is theoretically more efficient, the difference in practical terms is less clear. Typical sentences in a typical grammar (even a grammar as ambiguous as ROSIE's) will probably never realize the $O(n^{\delta+1})$ time bound. The variation on Tomita's algorithm distributed with ROSIE was implemented before the development of the modified algorithm. Minimal empirical results show that this implementation runs on the order of $n \log n$ for sentences ranging in length from one to several thousand symbols. It is unlikely that a similar implementation of the modified algorithm would perform substantially better. Thus, rather than accepting the modified algorithm as an "improved" algorithm for practical purposes, it should instead be treated as merely a proof that Tomita's algorithm can belong to the $O(n^3)$ class, illustrating its place in regards to the best-known general context-free algorithms.

REFERENCES

Aho, A.V., J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

Chomsky, N., "Three Models for the Description of Language," in *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113-124, 1956.

Chomsky, N., "On Certain Formal Properties of Grammars," in *Information and Control*, vol. 2, no. 2, pp. 137-167, 1959.

Earley, J., *An Efficient Context-Free Parsing Algorithm*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA, 1968.

Earley, J., "An Efficient Context-Free Parsing Algorithm," in *Communications of the ACM*, vol. 13, no. 2, pp. 94-102, Feb., 1970.

Hopcraft, J.E., J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

Kipps, J.R., B. Florman, H.A. Sowizral, *The New ROSIE Reference Manual and User's Guide*, R-3448-DARPA, The RAND Corporation, 1987.

Kipps, J.R., "A Table-Driven Approach to Fast Context-Free Parsing," N-2841-DARPA, The RAND Corporation, 1988.

Knuth, D.E., "On the Translation of Languages from Left to Right," *Information and Control*, vol. 8, pp. 607-639, 1965.

Tomita, M., *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA, 1985a.

Tomita, M., "An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications," in *Proceedings of the Ninth IJCAI*, vol. 2, pp. 756-764, Los Angeles, CA, Aug., 1985b.

Younger, D.H., "Recognition and Parsing of Context-Free Languages in Time n^3," in *Information and Control*, vol. 10, no. 2, pp. 189-208, 1967.